An Application of Deep Reinforcement Learning to Portfolio Management

By Wenzhao Wu

Abstracts
This essay presents an application of model-free reinforcement learning framework with deep neural networks for portfolio management. Two types of deep neural networks are used: Convolutional Neural Networks(CNN) and Long Short-Term Memory LSTM. Empirical Experiments are conducted to show that the reinforcement learning method with both neural networks constructed portfolios beat the market for most of the testing periods.

Keywords
Reinforcement Learning, Deep Neural Networks, CNN, LSTM, Portfolio Management

1. Introduction

Portfolio management is a process of building a portfolio and reallocating investment into different types of financial assets with the purpose of maximizing the portfolio return while constraining the risk.

There are many studies that have been performed on the application of deep learning or machine learning techniques to portfolio management problems. Predicting stock returns has been a widely discussed topic in both financial industry and academia. [1] For example, K. Nakagawa, T. Uchida, and T. Aoshima have tried to predict stock returns with different types of factors such as risk factors and momentum factors. [2] The idea of predicting stock returns with historical prices or indicators is easy to implement and it can often be reduced to regression problems. However, future asset prices are difficult to predict and predicting price does not help us in the construction of portfolios.

There are other applications of machine-learning schemes on algorithm trading problems that do not focus on predicting stock returns, but treat the problem as a reinforcement learning one. James Cumming has investigated the application of reinforcement learning to algorithm trading. [3] But he only investigated the possibility of trading one single asset and his work is not applicable to portfolio management problems where multiple assets are handled at the same time.

In this essay, I propose a deep reinforcement learning framework for the portfolio management. Unlike the traditional machine learning techniques, this framework does not aim at predicting asset prices. Instead, it hints at what actions should be taken to construct the portfolio in order to achieve a pre-specified goal. Under this framework, instead of using different indicators or factors to predict asset prices, it is assumed that all information is already reflected in the prices and we can just use the prices to decide on the allocation of different assets.

The idea of the framework is to construct a multi-asset portfolio using deep reinforcement learning framework. As a fully-machine learning approach, this framework should not be restricted to a specific industry. So the portfolio comprises of stocks of companies from different industries. This framework takes the historical prices of the stocks as an input and outputs the actions of allocating the weights of different assets through the whole portfolio management process. The core of this framework is to assign the weights of different assets and it is determined by two deep neural networks: Convolutional Neural Networks(CNN) and Long Short-Term Memory (LSTM). Within every trading period, the reinforcement learning agent, i.e. the portfolio manager receives the price information from the financial market, inputs it into the neural networks and reallocates the weights of assets according to the output of neural networks. The weights of different periods are stored in portfolio vector memory and updated through training process. The goal is to maximize the accumulative return through the portfolio management process and it is realized by the deterministic policy gradient method in the reinforcement learning process. In order to test the performance of deep reinforcement learning framework, five back-tests of different periods are performed. The performance of deep reinforcement learning is compared with a S&P500 index and an equal-weighted strategy. The performance of constructed portfolios is evaluated in terms of three metrics: accumulative return, Sharpe ratio and maximum drawdown. The empirical results show that both neural networks construct portfolios that can beat the market for most testing periods.

The essay is organized as follows: Section 2 will introduce the setup for the application of the deep reinforcement learning on the portfolio management problem. It will discuss the basics of the portfolio management and the technique of the reinforcement learning. In section 3, details are provided on how to incorporate deep neural networks into reinforcement learning. Two deep neural networks Convolutional Neural Networks(CNN) and Long Short-Term Memory (LSTM) will be presented. In section 4, empirical tests of the two deep reinforcement learning methods will be performed. An equal weighted portfolio and a S&P500 index will be also be compared with the portfolios constructed using the deep learning methods.

2. Problem Setup

2.1 Portfolio Management problem

A portfolio can be constructed with different types of financial assets such as stocks, bonds, commodities etc. Portfolio management is a process of constructing a portfolio by allocating the weights of different assets in a portfolio to achieve the goal of maximizing the return and minimize the risk. In this section, I will discuss how to construct a portfolio for a pre-determined trading period through the reinforcement deep learning framework.

2.2 Terminology of Reinforcement Learning

Reinforcement learning is a type of machine learning techniques which focuses on how an agent takes an action under a certain environment in order to achieve the goal of maximizing a pre-specified cumulative reward.

Within the reinforcement learning framework, there are several important items:

State: which represents the characteristics of current environment
Action: which is a decision made by agent
Reward: which is a consequence of an action
Policy: which comprises a set of rules used to determine what action to take

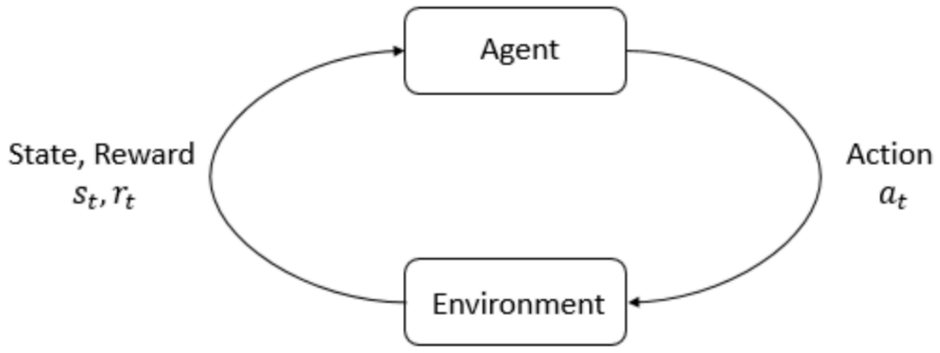

Figure1. reinforcement learning process

The figure above shows the reinforcement learning process of how the agent interacts with the environment. [4] Within one round of learning, the agent observes the state and takes actions based on that. The environment is changing all the time and can also be impacted by the agent's action. Then the agent will receive a feedback which is the reward from the environment indicating the result of the action and it will be reflected in the state as well. The agent keeps learning through multiple rounds of the process and eventually achieves the goal of maximizing a cumulative reward.

2.3 Fit Reinforcement Learning to Portfolio Management

In the portfolio management problem, the agent is a portfolio manager or a trader who constructs portfolios in the financial markets. The environment in which the agent faces reflects the information contained in the financial markets. It is believed by the technical trader that all of the information publicly available to the agents has already been reflected in the assets' price (Charles et al., 2006). With such as view, we can use historical price up to the current moment to represent the current state of the reinforcement learning process. Since we are trying to construct a portfolio, in addition to the assets price, weights of different assets also determine the value of the portfolio. As a result, we define the state as time t as combination of a price tensor X_t and a weight vector ω_{t-1}:

$$s_t = (X_t,\ \omega_{t-1})$$

The action that agent takes at time t is ω_t which reflects how the agent plans to allocate weights to different assets at time t:

$$a_t = \omega_t$$

There are two important assumptions which need to be made under this reinforcement learning framework:

1. All selected financial assets are liquid enough that the trades can be executed immediately with the required volume at the last price whenever the order is placed.

2. The trade action will not have any impact on the markets. So it means that reallocating the weights of the portfolio will not have any impact on the environment or the next day's prices.

The immediate reward of one round of learning is the day by day return of the portfolio after reallocating the portfolio weights. The goal of the agent is to maximize the accumulative reward at the end of the portfolio management process.

The action of the framework is determined by a policy π based on state:

$$\omega_t = a_t = \pi(s_t)$$

Under our deep reinforcement learning framework, policy function $\pi(s_t)$ is constructed by two types of deep neural networks which will be discussed in detail in next section.

3. Neural Networks

3.1 Input and Output of Neural Networks

Assume there are m assets selected to construct the portfolio and the portfolio management process last for k trading periods which is k trading days under our setup.

N days of historical prices of the assets are inputs into the neural networks to generate the portfolio weights. At each time t, the price tensor X_t is an input into the neural networks and an output of the neural networks is denoted by ω_t. The input price tensor at time t is X_t defined as follows:

$X_t = [V_t, V_t^{(lo)}, V_t^{(hi)}, V_t^{(op)}]$
where
V_t represents close prices and $V_t = [v_{t-n+1} \varnothing v_t \mid v_{t-n+2} \varnothing v_t \mid \ldots \mid \mathbf{1}]$
$V_t^{(lo)}$ represents lowest prices and $V_t^{(lo)} = [\, v_{t-n+1}^{(lo)} \varnothing v_t \mid v_{t-n+2}^{(lo)} \varnothing v_t \mid \ldots \mid \mathbf{1}]$
$V_t^{(hi)}$ represents highest prices and $V_t^{(hi)} = [\, v_{t-n+1}^{(hi)} \varnothing v_t \mid v_{t-n+2}^{(hi)} \varnothing v_t \mid \ldots \mid \mathbf{1}]$
$V_t^{(op)}$ represents open prices and $V_t^{(op)} = [\, v_{t-n+1}^{(op)} \varnothing v_t \mid v_{t-n+2}^{(op)} \varnothing v_t \mid \ldots \mid \mathbf{1}]$
and
$v_{t-1} \varnothing v_t = (1, \frac{v_{1,t-1}}{v_{1,t}}, \frac{v_{2,t-1}}{v_{2,t}}, \ldots, \frac{v_{m,t-1}}{v_{m,t}})^T$

The prices are normalized by the latest price because only the change in prices will have an impact on the agent's decision, and not the absolute value of the prices. And normalization helps to improve training efficiency.

3.2 Convolutional Neural Network(CNN)

Convolution is a specialized type of a linear operation. So convolutional neural networks are neural networks that use a convolution instead of a general matrix multiplication inside of at least one of its layers. A traditional application of CNN in finance is trying to predict the stock prices. However, here instead of predicting stock prices, we are trying to predict the weights of the portfolio in order to achieve the optimization goal.

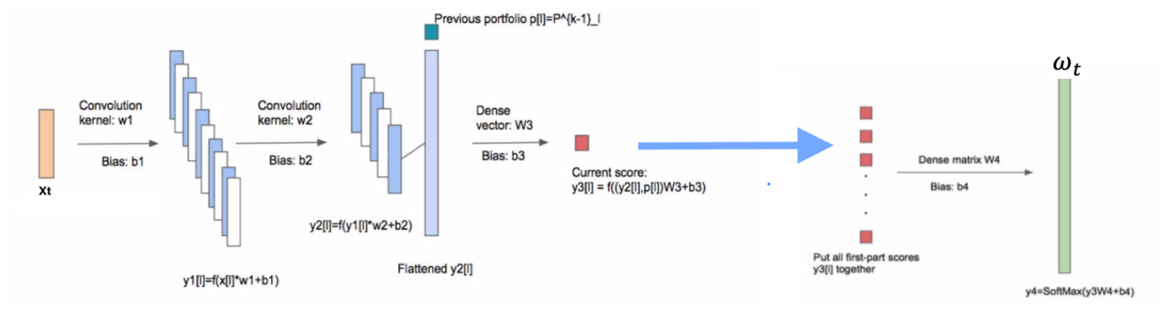

Figure 2. CNN framework

From the figure 2 [5] above we can see that each assets' prices are input into the neural networks. The last hidden layer of the neural networks assigns a voting score on the growth potential of the assets. The information of m assets goes through the hidden layers separately and interacts at the last output layer through a softmax function. The softmax function takes the voting score of the assets and outputs the portfolio weights vector ω_t.

The convolutional neural networks are constructed with one input layer, three hidden layers and one output layer. The input layer takes the input tensor Xt which is in a shape of (4, M, N). 4 represents four features of prices: last price, open price, highest price, lowest price. M represents the number of assets and N represents the length of each stock's historical price. Each layer in the neural network creates an output value by applying a specific function f(•) to the input values coming from previous layer. The output value is determined by a vector of weights w_i and bias b_i and the function f(•). The vector of weights w_i and bias b_i represents the features of input from previous layer and is iteratively adjusted through the training process. Function tanh is used in the first and second layer and function ReLu is used in the third layer. Both tanh and ReLu are non-linear operations that have been proven to have best performance under most situations. A voting score will be created for each asset after going through the three convolutional layers. And the voting score of all assets will be put together and normalized by the activation function softmax so that the output weights in ω_t sum up to 1. The detailed values of hyper-parameters of CNN used in the experiments can be found in appendix.

3.3 Long Short-Term Memory (LSTM)

Long Short-Term Memory is a type of recurrent neural network(RNN) applied in the deep learning field. Different from the standard feedforward neural networks, the nodes in LSTM do not only take in inputs from the previous layer, but also provide feedbacks to the previous nodes. LSTM is often used to extract information from sequences of data. Below is an illustration of LSTM memory blocks from which we can see how the feedback connections work in a LSTM memory blocks. [6] The forget gate, the input gate and the output gate at step t are represented by the letters f_t, i_t, o_t respectively. The forget gate f_t decides whether to keep or dispose of a certain piece of information from the previous cell, the input gate i_t decides which information we are going to update, and the output gate o_t passes the filtered information to the next cell.

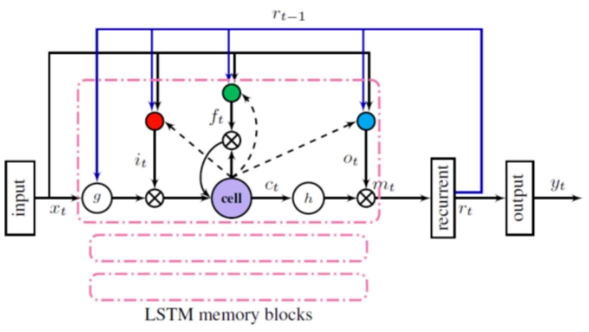

Figure2. LSTM memory blocks

If we unfold the memory blocks in figure2 we will get the middle part of figure3. LSTM can process the entire sequence of data. The cell keeps tracks of the dependencies among the elements in the input sequence. In our portfolio management problem, the setup of LSTM is very similar to CNN. The input to the neural networks is the historical prices of individual asset. The important information is extracted through the layers of LSTM and each asset is assigned with a voting score at last hidden layer and concatenate with other assets through the softmax function and outputs the portfolio weights. Below is an example of the implementation of LSTM on constructing a portfolio.

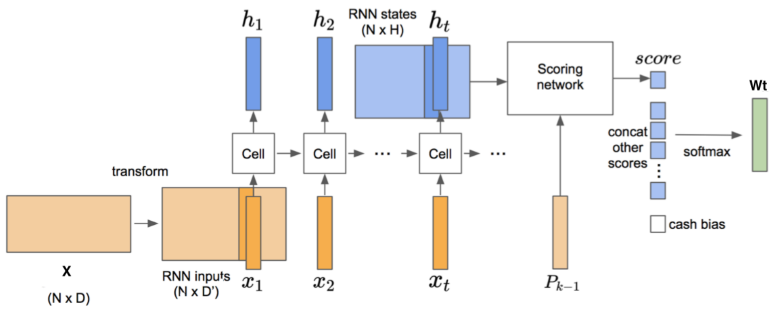

Figure3. LSTM process

3.4 Reward Function

From neural networks we can generate a sequence of weights vectors $\{\omega_1, \omega_2, \ldots, \omega_k\}$ for the entire portfolio management process. In the portfolio management, another problem we need to consider is the transaction cost when reallocating the weights of assets in the portfolio. [7] [8] The reward function or the objection function we would like to maximize, denoted as R_k can be expressed as:

$R_k = \sum_{t=1}^{k} \ln(\omega_t * y_t * (1 - C * \sum_{i=1}^{m} |\omega_{t-1,i} - \omega_{t,i}|))$
where
$y_t = v_t \varnothing v_{t-1} = (1, \frac{v_{1,t}}{v_{1,t-1}}, \frac{v_{2,t}}{v_{2,t-1}}, \ldots, \frac{v_{m,t}}{v_{m,t-1}})^T$ which is the daily return
C is the transaction cost

3.5 Portfolio Vector Memory

In order to restrain the transaction cost from shifting the weights considerably between consecutive portfolio weight vectors, the output of portfolio weights from the previous trading period is used as an input to next networks to take the cash bias into account. The whole framework is presented in the following figure. [9] Figure (a) represents the training scheme of a mini-batch. In figure (b), at each step of the training, the weights of previous trading period are read into the neural networks, and the output of the portfolio vector from the neural networks is then placed to the portfolio vectors memory.

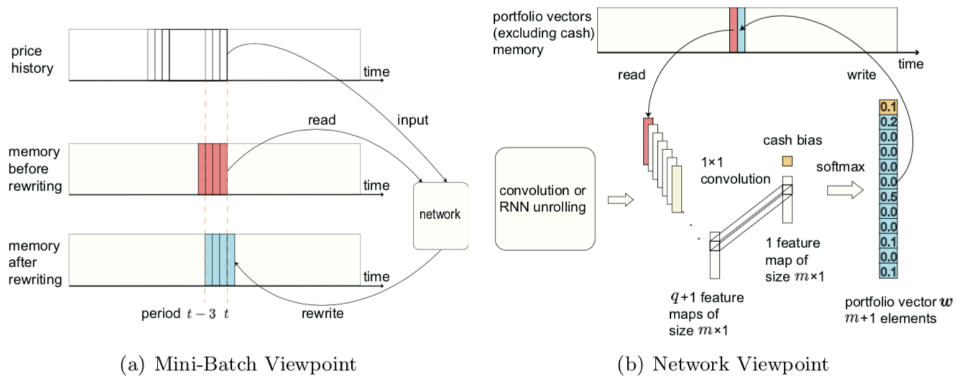

Figure 4. Portfolio Vector Memory

3.6 Deterministic Policy Gradient

Policy gradient algorithm optimizes a policy by computing noisy estimates of the gradient of the expected reward of the policy and then updating the policy in the gradient direction. [10] Assume a policy is specified by a set of parameters θ such that $a_t = \pi_\theta(s_t)$. The performance metric of policy is defined as the reward function R_k:

$$J_{[0,k]}(\pi_\theta) = R(s_1, \pi_\theta(s_1), s_2, \pi_\theta(s_2), \ldots, s_k, \pi_\theta(s_k))$$

After given initial values, the set of parameters θ is updated in the gradient direction with a learning rate λ:

$$\theta \rightarrow \theta + \lambda \nabla J_{[0,k]}(\pi_\theta)$$

Under current framework, in order to improve training efficiency, mini-batch scheme is applied. The length of updating parameters is reduced to a shorter period [t_1, t_2]. The parameters will be updated upon mini-batches instead of the whole training period [0, k]:

$$\theta \rightarrow \theta + \lambda \nabla J_{[\mathrm{t1,t2}]}(\pi_\theta)$$

The training process is to find the optimal policy using policy gradient algorithm to maximize the reward function R_k. The optimization process is realized by Adaptive Moment Estimation (Adam) method which can be easily implemented with TensorFlow in Python. [11] The values of hyper-parameters of the optimizer are listed in the Appendix table.

4. Experiments

To construct the portfolio, we select seven stocks from different industries. They are stocks of Apple Inc., Microsoft Corporation, Boeing Co, Amazon, General Electric Company, Google, JPMorgan Chase & Co. Those stocks all have high trading volumes, such that they satisfy the previously stated two assumptions that the assets are liquid enough and our trading actions will have little influence on the financial market.

The reinforcement deep learning result is examined under 5 back-tests across 10 years from April 6th, 2010 to April 6th 2020. Each back-test contains data of approximately two years. The training, validating and testing dataset within each back-test are respectively distributed with 60%, 20%, 20% of the entire dataset. Aside from the portfolios constructed by using the reinforcement deep learning methods, an equally weighted portfolio is also constructed as benchmark. In addition, the S&P500 index is also used for comparison to show whether our portfolios can beat the market. The periods of the data for each test is presented in the table below:

	Data Period
Backtest 1	2010-04-07 to 2011-08-31
Backtest 2	2013-01-02 to 2014-12-31
Backtest 3	2015-01-02 to 2016-12-30
Backtest 4	2017-01-03 to 2018-12-31
Backtest 5	2019-01-02 to 2020-04-06

Table1. Data Period

4.1 Performance Measures

There are three measures that are used to evaluate the performance of the constructed portfolios. The first one is the accumulative portfolio return(APR). Here we assume an initial investment of p_0 = 10000 dollars at the beginning of the portfolio management process. Then APR = p_k / p_0 – 1 for a process of k days.

The second metric is a Sharpe Ratio of the daily return of the portfolio management process. Comparing to the APR which only considers the return, the Sharpe Ratio takes risk into account as well.

The third measure is Max Drawdown which tests the downside deviation of the portfolio. It is calculated as the percentage difference from peak to trough.

4.2 Experiments Results

The results of the five back-tests are presented below. From the performance of the different portfolios under the five back-tests we can say that CNN performs the best for most of the time in terms of accumulative portfolio return, Sharpe ratio and Maximum Drawdown. Both CNN and LSTM help to construct the portfolios that have a higher accumulative return than that from S&P500 index. There is one special case, under the back-test 3, where CNN gives the worst result among the four methods. Another thing worth mentioning is that under back-test5 which is the most recent training period, the market has dropped a lot during to rising risk from COVID-19, CNN still gives a 10.49% accumulative return compared to a 20.29% drop in the S&P 500 index. Another observation is although the deep reinforcement learning network outperforms the market in terms of return, its returns are not very stable. Both CNN and LSTM have low Sharpe Ratios and sometimes they bring in a relatively severe drawdown.

Back-test 1	APR	SR	MaxDrawdown
EW	11.55%	0.2133	2.60%
S&P500	-9.39%	-0.0701	16.77%
CNN	16.13%	0.1581	9.77%
LSTM	12.80%	0.1785	6.18%

Table 2. Performance under Back-test1

Back-test 2	APR	SR	MaxDrawdown
EW	8.87%	0.2477	2.03%
S&P	4.10%	0.0739	13.22%
CNN	6.03%	0.1236	2.60%
LSTM	13.93%	0.2149	4.12%

Table 3. Performance under Back-test2

Back-test 3	APR	SR	MaxDrawdown
EW	2.76%	0.0342	6.91%
S&P	2.79%	0.0499	3.08%
CNN	-3.42%	-0.0220	13.37%
LSTM	8.99%	0.0597	12.94%

Table 4. Performance under Back-test3

Back-test 4	APR	SR	MaxDrawdown
EW	5.77%	0.0933	3.60%
S&P	-11.73%	-0.0992	15.75%
CNN	16.93%	0.1873	2.65%
LSTM	6.03%	0.0533	8.38%

Table 5. Performance under Back-test4

Back-test 5	APR	SR	MaxDrawdown
EW	3.73%	0.0635	5.21%
S&P	-20.29%	-0.0657	3.73%
CNN	10.49%	0.0926	8.88%
LSTM	0.45%	0.0096	5.90%

Table 6. Performance under Back-test5

Below is an example of the portfolio value changing process of back-test 5 under different methods. It shows how the deep reinforcement learning architecture maintains the value of the portfolio under unexpected events that impact the entire markets.

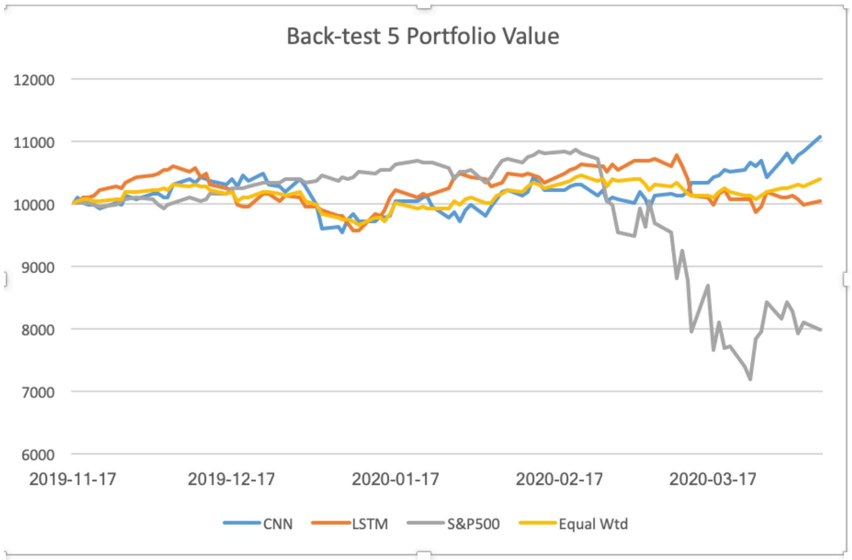

Figure5. Back-test 5 Portfolio Value

Another thing worth mentioning is that the CNN method is more efficient than the LSTM method. Taking Back-test 5 for example, with the same size of dataset, it costs the CNN method 866 seconds to complete the training and testing process while the LSTM method spends 4321 seconds to complete the whole process.

5. Summary

Based on the experiments results we may conclude that the deep reinforcement learning method helps to construct portfolios that can outperform the market for most of the time. Also, it helps to automate the portfolio management process and improve work efficiency. However, the deep reinforcement learning method also brings in uncertainty to the construction of portfolios. And the assumptions that the market is liquid enough and agents' action has no impact on the market might not be realistic. There are still some improvements that can be made to take agents' action impact and market liquidity into consideration. Also, it is worthwhile to try this method by using different types of assets and test for different markets. In addition, we can should test using different hyper-parameters of the models to see which one fit best for certain assets and markets.

Appendix A. Hyper-parameters

Hyper-Parameters	Value
Batch Size	40
Number of Assets	7
Regularization Coefficient	1 x 10^-8
Learning Rate	9 x 10^-2
Commision Rate	1 x 10^-6
Trading Period	10
Number of Episodes	4
Initial Portfolio Value	10000
CNN kernel size	(1,3)
CNN layer 1 filter	2
CNN layer 2 filter	20

Appendix B. Python Script

[6]:
```
!pip install nbconvert

import pandas as pd
import numpy as np
```

[49]:
```
stocks = pd.read_csv("~/Desktop/MMATH/Research/data/test5.csv")
```

[50]:
```
stocks_data  = stocks.sort_values(['ticker', 'Date'])
```

[51]:
```
print('stocks considered : ' + str(np.unique(stocks_data['ticker'])))
```

```
stocks considered : ['AAPL' 'AMZN' 'BA' 'GE' 'GOOG' 'JPM' 'MSFT']
```

[52]:
```
pivot_table = pd.pivot_table(stocks_data, values='Close',␣
 ↪index=['Date'],columns=['ticker'])
```

[53]:
```
returns = pivot_table.pct_change().dropna()
stocks_returns_mean = returns.mean()
stocks_covariance_matrix = returns.cov()
```

[54]:
```
def get_input_data(data_table, ticker_column_name):
    open_values = list()
    close_values = list()
    high_values = list()
    low_values = list()

    for ticker in np.unique(data_table[ticker_column_name]):
        open_value_list = data_table['Open'].
 ↪loc[data_table[ticker_column_name]==ticker]
        open_values.append(open_value_list[1:].reset_index(drop = True) /␣
 ↪open_value_list[:-1].reset_index(drop = True))
        close_values.append(data_table['Close'].
 ↪loc[data_table[ticker_column_name]==ticker][:-1] / open_value_list[:-1])
        high_values.append(data_table['High'].
 ↪loc[data_table[ticker_column_name]==ticker][:-1] / open_value_list[:-1])
        low_values.append(data_table['Low'].
 ↪loc[data_table[ticker_column_name]==ticker][:-1] / open_value_list[:-1])
```

```
    return np.array([close_values,
                     high_values,
                     low_values,
                     open_values])
```

[55]:
```
stocks_input_array = get_input_data(stocks_data, 'ticker')
np.save('stocks_data_input.npy', stocks_input_array)
print('shape of stocks data input : ' + str(stocks_input_array.shape))
```

```
shape of stocks data input : (4, 7, 317)
```

[4]:
```
!pip install tensorflow
```

[5]:
```
import tensorflow as tf
```

[56]:
```
#dataset information
data_path = 'stocks_data_input.npy'
ticker_list = ['AAPL', 'BA', 'AMZN','JPM', 'GE', 'MSFT', 'GOOG']
```

[57]:
```
data = np.load(data_path)
ohlc_features_num = data.shape[0]
ticker_num = data.shape[1]
trading_days_captured = data.shape[2]
```

[58]:
```
print('number of ohlc features : ' + str(ohlc_features_num))
print('number of stocks considered : ' + str(ticker_num))
print('number of trading days captured : ' + str(trading_days_captured))

equiweight_weights_stocks = np.array([np.array([1/(ticker_num)]*(ticker_num+1))])
```

```
number of ohlc features : 4
number of stocks considered : 7
number of trading days captured : 317
```

[59]:
```
from stocks_parameters import *
```

[60]:
```
import tensorflow as tf
import tqdm
import numpy as np

from RLEnvironment import *
from policy_cnn import *
from pf_vector_memory import *
from stocks_parameters import *
from data_pre_processing import *
```

[61]:
```
def get_random_action(num_tickers):
    vector_rand = np.random.rand(num_tickers + 1)
    return vector_rand / np.sum(vector_rand)
```

[62]:
```
def max_drawdown(weights, time_period=num_trading_periods,␣
 ↪portfolio_data_frame=pivot_table):
    weights = weights.reshape(len(weights[0],))
    weights = weights[1:]
    simulated_portfolio = weights[0]*portfolio_data_frame.iloc[:,0]  #changed␣
 ↪here from .ix to .iloc, if not working try .loc
    for i in range(1, len(portfolio_data_frame.columns)):
        simulated_portfolio += weights[i]*portfolio_data_frame.iloc[:,i]␣
 ↪#changed here from .ix to .iloc, if not working try .loc
    max_drawdown_value = float('-inf')
    for i in range(int(len(simulated_portfolio)/time_period)-1):
        if min(simulated_portfolio[i*time_period:(i+1)*time_period]) > 0:
            biggest_variation = max(simulated_portfolio[i*time_period:
 ↪(i+1)*time_period])/min(simulated_portfolio[i*time_period:(i+1)*time_period])
        else:
            biggest_variation = 0
        if(biggest_variation > max_drawdown_value):
            max_drawdown_value = biggest_variation
    return max_drawdown_value
```

[63]:
```
def RoMad(weights, mdd,␣
 ↪returns_mean=stocks_returns_mean,time_period=num_trading_periods):
    weights = weights.reshape(len(weights[0],))
    weights = weights[1:]
    portfolio_return = np.sum(returns_mean.values * weights * time_period)
    if mdd>0:
        romad = portfolio_return/mdd
    else:
        romad=0
    return romad
```

[64]:
```
def sharpe_stocks(w ,returns_cov=stocks_covariance_matrix,␣
 ↪returns_mean=stocks_returns_mean ):
    w = w.reshape(len(w[0],))
    w = w[1:]
    portfolio_return = np.sum(returns_mean.values * w * nbr_trading_days)
    portfolio_volatility = np.sqrt(np.dot(w.T, np.dot(returns_cov.values, w))*␣
 ↪nbr_trading_days)
    sharpe_ratio = portfolio_return / portfolio_volatility
    return sharpe_ratio
```

[65]:
```
sharpe_equiweight = round(sharpe_stocks(w=equiweight_weights_stocks), 3)
```

[66]:
```
mdd_equiweight = round(max_drawdown(weights=equiweight_weights_stocks), 3)
```

[2]:
```
!pip install --upgrade tensorflow
!pip install --upgrade keras
```

[3]:
```
!pip install tensorflow==1.13.1
```

[68]:
```
import matplotlib.pyplot as plt
import numpy as np

#function to plot cumulative portfolio values
def plot_cpv(final_pf_values_opt, final_pf_values_eq, initial_portfolio_value,␣
 ↪plot_name):
        plt.plot(final_pf_values_opt, label = 'optimizing agent')
        plt.plot(final_pf_values_eq, label = 'equiweight agent')
        plt.plot([initial_portfolio_value] * len(final_pf_values_opt), label =␣
 ↪'no investment')
        plt.legend()
        plt.title('cumulative portfolio values over test steps')
        plt.xlabel('test steps')
        plt.ylabel('cumulative portfolio value')
        plt.savefig('figures/test_result_plots/' + plot_name)

#function to plot final wt vectors assigned
def plot_wts_assigned(wt_vector, num_stocks, ticker_list, plot_name):
        plt.bar(np.arange(num_stocks + 1), wt_vector)
        plt.title('Final Portfolio weights (test set)')
        plt.xticks(np.arange(num_stocks + 1), ['Cash'] + ticker_list,␣
 ↪rotation=45)
        plt.xlabel('tickers')
        plt.savefig('figures/test_result_plots/' + plot_name)
```

[8]:
```
#import tensorflow as tf
import timeit
tic=timeit.default_timer()



#import tensorflow.compat.v1 as tf
#tf.disable_v2_behavior()
import tensorflow as tf

from tensorflow import keras
from tensorflow.python.keras import backend as k




def main(stocks = True):
        #environment set up for the portfolio optimizing agent
        env_pf_optimizer = RLEnv(Path = data_path, PortfolioValue =␣
↪portfolio_value_init, TransCost = trading_cost,
                                    ReturnRate = interest_rate, WindowSize =␣
↪num_trading_periods, TrainTestSplit = train_data_ratio)

        #environment set up for the portfolio equiweight agent
        env_pf_equiweight = RLEnv(Path = data_path, PortfolioValue =␣
↪portfolio_value_init, TransCost = trading_cost,
                                    ReturnRate = interest_rate, WindowSize =␣
↪num_trading_periods, TrainTestSplit = train_data_ratio)

#---------------------------------------- training - Using the RL framework␣
↪------------------------------------
        #tf.reset_default_graph()
        tf.reset_default_graph()

        with tf.Session() as sess:

                # initialize networks
            pf_opt_agent = PolicyCNN(ohlc_features_num, ticker_num,␣
↪num_trading_periods, sess, optimizer, trading_cost, cash_bias_init,␣
↪interest_rate,
                                        equiweight_vector,␣
↪adjusted_rewards_alpha, kernel_size, num_filters_layer_1, num_filters_layer_2)

                # initialize tensorflow graphs
            sess.run(tf.global_variables_initializer())

            train_pf_values = []
            train_pf_values_equiweight = []

            for ep_num in tqdm.tnrange(num_episodes, desc = 'Episodes'):
                        #Creating PVm memory Object
                pvm = PFVectorMemory(ticker_num, beta_pvm, training_steps,␣
↪training_batch_size, weight_vector_init)

                for batch_num in tqdm.tnrange(num_batches, desc = 'Batches'):
                        list_X_t, list_wt_previous, list_pf_value_previous,␣
↪list_daily_returns_t, list_pf_value_previous_equiweight, sharpe_ratio_train,␣
↪mdd_train = [], [], [], [], [], [], []

                                #Selecting the first time t
                        training_batch_t_selection = False
                        while training_batch_t_selection == False:
                                training_batch_t = training_steps -␣
↪training_batch_size + 1 - np.random.geometric(p = beta_pvm)
                                if training_batch_t >= 0:
                                        training_batch_t_selection = True

                        state, done = env_pf_optimizer.ResetEnvironment(pvm.
↪get_wt_vector_t(int(training_batch_t)), portfolio_value_init ,␣
↪training_batch_t  )
                        state_equiweight, done_equiweight = env_pf_equiweight.
↪ResetEnvironment(equiweight_vector, portfolio_value_init, training_batch_t)

                        for training_batch_num in tqdm.
↪tnrange(training_batch_size, desc = 'Training Batches'):

                                X_t = state[0].reshape([-1] + list(state[0].
↪shape))
                                Wt_previous = state[1].reshape([-1] +␣
↪list(state[1].shape))
                                pf_value_previous = state[2]

                                action = pf_opt_agent.compute_weights(X_t,␣
↪Wt_previous) if (np.random.rand() < epsilon) else get_random_action(ticker_num)
                                state, reward, done = env_pf_optimizer.
↪Step(action)
                                state_equiweight, reward_equiweight,␣
↪done_equiweight = env_pf_equiweight.Step(equiweight_vector)

                                X_next = state[0]
                                Wt_t = state[1]
                                pf_value_t = state[2]
                                pf_value_t_equiweight = state_equiweight[2]

                                daily_returns_t = X_next[-1, :, -1]
                                pvm.update_wt_vector_t(training_batch_t +␣
↪training_batch_num, Wt_t)

                                list_X_t.append(X_t.reshape(state[0].shape))
                                list_wt_previous.append(Wt_previous.
↪reshape(state[1].shape))
                                list_pf_value_previous.
↪append([pf_value_previous])
                                list_daily_returns_t.append(daily_returns_t)
                                list_pf_value_previous_equiweight.
↪append(pf_value_t_equiweight)
```

```
                                sharpe_ratio =␣
↪round(sharpe_stocks(w=Wt_previous), 3)
                                mdd = round(max_drawdown(weights=Wt_previous), 3)
                                romad = round(RoMad(weights=Wt_previous,␣
↪mdd=mdd))
                                # print('------------------ training␣
↪-----------------------')
                                # print('current portfolio value : ' +␣
↪str(pf_value_previous))
                                print('weights assigned : ' + str(Wt_t))
                                print('sharpe_ratio:', sharpe_ratio)
                                print('RoMaD', romad)
                                # print('equiweight portfolio value : ' +␣
↪str(pf_value_t_equiweight))
                                # print("equiweight sharpe", sharpe_equiweight)

                        train_pf_values.append(pf_value_t)
                        sharpe_ratio_train.append(sharpe_ratio)
                        train_pf_values_equiweight.append(pf_value_t_equiweight)
                        mdd_train.append(romad)

                        #training the network after each batch to maximize the␣
↪reward
                        pf_opt_agent.train_cnn(np.array(list_X_t),
                                               np.array(list_wt_previous),
                                               np.array(list_pf_value_previous),
                                               np.array(list_daily_returns_t))

                # Add ops to save and restore all the variables.
                saver = tf.train.Saver()
                # Save the variables to disk.
                save_path = saver.save(sess, "/tmp/model.ckpt")
                print("Model saved in path: %s" % save_path)

                print('------ train final value -------')
                print(train_pf_values)
                print(train_pf_values_equiweight)
                # print(sharpe_ratio_train)
        #---------------------------------------- testing of the trained␣
↪portfolio optimizer ------------------------------------

            state, done = env_pf_optimizer.
↪ResetEnvironment(weight_vector_init_test, portfolio_value_init_test,␣
↪training_batch_t)
```

```
            state_equiweight, done_equiweight = env_pf_equiweight.
↪ResetEnvironment(equiweight_vector, portfolio_value_init_test,␣
↪training_batch_t)

            test_pf_values = [portfolio_value_init_test]
            test_pf_values_equiweight = [portfolio_value_init_test]
            weight_vectors = [weight_vector_init_test]
            test_sharpe_ratio = []
            test_mdd = []

            start_step_num = int(training_steps  + validation_steps -␣
↪int(num_trading_periods/2))
            end_step_num = int(training_steps  + validation_steps + test_steps -␣
↪num_trading_periods)

            for test_step_num in range(start_step_num, end_step_num):

                X_t = state[0].reshape([-1] + list(state[0].shape))
                wt_previous = state[1].reshape([-1] + list(state[1].shape))
                pf_value_previous = state[2]

                action = pf_opt_agent.compute_weights(X_t, wt_previous)
                state, reward, done = env_pf_optimizer.Step(action)
                state_equiweight, reward_equiweight, done_equiweight =␣
↪env_pf_equiweight.Step(equiweight_vector)

                X_next_t, wt_t, pf_value_t = state[0], state[1], state[2]
                pf_value_t_equiweight = state_equiweight[2]
                daily_returns_t = X_next_t[-1, :, -1]
                sharpe_ratio = round(sharpe_stocks(w=wt_previous),3)
                mdd = round(max_drawdown(weights=Wt_previous), 3)
                romad = round(RoMad(weights=Wt_previous, mdd=mdd))
                # print('------------------ testing -----------------------')
                # print('current portfolio value : ' + str(pf_value_previous))
                print('weights assigned : ' + str(wt_previous))
                print('sharpe_ratio:', sharpe_ratio)
                print('RoMaD', romad)
                # print('equiweight portfolio value : ' +␣
↪str(pf_value_t_equiweight))
                # print("equiweight sharpe", sharpe_equiweight)

                test_pf_values.append(pf_value_t)
                test_pf_values_equiweight.append(pf_value_t_equiweight)
                weight_vectors.append(wt_t)
                test_sharpe_ratio.append(sharpe_ratio)
                test_mdd.append(romad)
            print('------ test final value -------')
            print(test_pf_values)
            print(test_pf_values_equiweight)
            print(np.mean(test_sharpe_ratio))
            print(np.mean(test_mdd))
            #plot_cpv(test_pf_values, test_pf_values_equiweight,␣
 ↪portfolio_value_init_test, 'cpv_stocks.png')
            #plot_wts_assigned(weight_vectors[-1], ticker_num, ticker_list,␣
 ↪'wt_vector_stocks.png')


main()

toc=timeit.default_timer()
toc - tic #elapsed time in seconds
```

[8]:
```
0.0036708699995081
```

[9]:
```
!pip install tflearn
```

[14]:
```
from policy_lstm import *
num_filter_layer= 20




#print("Equiweighted Sharpe",sharpe_equiweight)
#print("Equiweighted MDD",mdd_equiweight)

tic=timeit.default_timer()

def main(stocks = True):
        #environment set up for the portfolio optimizing agent
        env_pf_optimizer = RLEnv(Path = data_path, PortfolioValue =␣
 ↪portfolio_value_init, TransCost = trading_cost,
                                   ReturnRate = interest_rate, WindowSize =␣
 ↪num_trading_periods, TrainTestSplit = train_data_ratio)

        #environment set up for the portfolio equiweight agent
        env_pf_equiweight = RLEnv(Path = data_path, PortfolioValue =␣
 ↪portfolio_value_init, TransCost = trading_cost,
                                   ReturnRate = interest_rate, WindowSize =␣
 ↪num_trading_periods, TrainTestSplit = train_data_ratio)

#---------------------------------------- training - Using the RL framework␣
 ↪------------------------------------
        tf.reset_default_graph()
```

```
    with tf.Session() as sess:

        # initialize networks
        pf_opt_agent = PolicyLSTM(ohlc_features_num, ticker_num,␣
↪num_trading_periods, sess, optimizer, trading_cost, cash_bias_init,␣
↪interest_rate, equiweight_vector, adjusted_rewards_alpha, num_filter_layer)

        # initialize tensorflow graphs
        sess.run(tf.global_variables_initializer())

        train_pf_values = []
        train_pf_values_equiweight = []

        for ep_num in tqdm.tnrange(num_episodes, desc = 'Episodes'):
            #Creating PVm memory Object
            pvm = PFVectorMemory(ticker_num, beta_pvm, training_steps,␣
↪training_batch_size, weight_vector_init)

            for batch_num in tqdm.tnrange(num_batches, desc = 'Batches'):
                list_X_t, list_wt_previous, list_pf_value_previous,␣
↪list_daily_returns_t, list_pf_value_previous_equiweight, sharpe_ratio_train =␣
↪[], [], [], [], [], []

                #Selecting the first time t
                training_batch_t_selection = False

                while training_batch_t_selection == False:
                    training_batch_t = training_steps - training_batch_size␣
↪+ 1 - np.random.geometric(p = beta_pvm)
                    if training_batch_t >= 0:
                        training_batch_t_selection = True

                state, done = env_pf_optimizer.ResetEnvironment(pvm.
↪get_wt_vector_t(int(training_batch_t)), portfolio_value_init ,␣
↪training_batch_t  )
                # print(state[0].reshape([-1] + list(state[0].shape)))
                state_equiweight, done_equiweight = env_pf_equiweight.
↪ResetEnvironment(equiweight_vector, portfolio_value_init, training_batch_t)

                for training_batch_num in tqdm.tnrange(training_batch_size,␣
↪desc = 'Training Batches'):

                    X_t = state[0].reshape([-1] + list(state[0].shape))
                    # print("32734874837438473847 83", X_t.shape)
```

```
                        Wt_previous = state[1].reshape([-1] + list(state[1].
↪shape))
                        pf_value_previous = state[2]

                        action = pf_opt_agent.compute_weights(X_t, Wt_previous)␣
↪if (np.random.rand() < epsilon) else get_random_action(ticker_num)
                        state, reward, done = env_pf_optimizer.Step(action)
                        state_equiweight, reward_equiweight, done_equiweight =␣
↪env_pf_equiweight.Step(equiweight_vector)

                        X_next = state[0]
                        Wt_t = state[1]
                        pf_value_t = state[2]
                        pf_value_t_equiweight = state_equiweight[2]

                        daily_returns_t = X_next[-1, :, -1]
                        pvm.update_wt_vector_t(training_batch_t +␣
↪training_batch_num, Wt_t)

                        list_X_t.append(X_t.reshape(state[0].shape))
                        list_wt_previous.append(Wt_previous.reshape(state[1].
↪shape))
                        list_pf_value_previous.append([pf_value_previous])
                        list_daily_returns_t.append(daily_returns_t)
                        list_pf_value_previous_equiweight.
↪append(pf_value_t_equiweight)
                        sharpe_ratio = round(sharpe_stocks(w=Wt_previous), 3)
                        mdd = round(max_drawdown(weights=Wt_previous), 3)
                        romad = round(RoMad(weights=Wt_previous, mdd=mdd))
                        # print('------------------ training␣
↪-----------------------')
                        # print('current portfolio value : ' +␣
↪str(pf_value_previous))
                        print('weights assigned : ' + str(Wt_t))
                        print('sharpe_ratio:', sharpe_ratio)
                        print('RoMad:', romad)
                        # print('equiweight portfolio value : ' +␣
↪str(pf_value_t_equiweight))
                        # print("equiweight sharpe", sharpe_equiweight)

                    train_pf_values.append(pf_value_t)
                    sharpe_ratio_train.append(sharpe_ratio)
                    train_pf_values_equiweight.append(pf_value_t_equiweight)

                    #training the network after each batch to maximize the reward
                    pf_opt_agent.train_lstm(np.array(list_X_t),
                        np.array(list_wt_previous),
                        np.array(list_pf_value_previous),
                        np.array(list_daily_returns_t))

                # Add ops to save and restore all the variables.
                saver = tf.train.Saver()
                # Save the variables to disk.
                save_path = saver.save(sess, "/tmp/model.ckpt")
                print("Model saved in path: %s" % save_path)

                print('------ train final value -------')
                print(train_pf_values)
                print(train_pf_values_equiweight)
                print(sharpe_ratio_train)
        #---------------------------------------- testing of the trained␣
↪portfolio optimizer ------------------------------------

            state, done = env_pf_optimizer.
↪ResetEnvironment(weight_vector_init_test, portfolio_value_init_test,␣
↪training_batch_t)
            state_equiweight, done_equiweight = env_pf_equiweight.
↪ResetEnvironment(equiweight_vector, portfolio_value_init_test,␣
↪training_batch_t)

            test_pf_values = [portfolio_value_init_test]
            test_pf_values_equiweight = [portfolio_value_init_test]
            weight_vectors = [weight_vector_init_test]
            test_sharpe_ratio = []
            test_mdd = []

            start_step_num = int(training_steps  + validation_steps -␣
↪int(num_trading_periods/2))
            end_step_num = int(training_steps  + validation_steps + test_steps -␣
↪num_trading_periods)

            for test_step_num in range(start_step_num, end_step_num):
                X_t = state[0].reshape([-1] + list(state[0].shape))
                wt_previous = state[1].reshape([-1] + list(state[1].shape))
                pf_value_previous = state[2]

                action = pf_opt_agent.compute_weights(X_t, wt_previous)
                state, reward, done = env_pf_optimizer.Step(action)
                state_equiweight, reward_equiweight, done_equiweight =␣
↪env_pf_equiweight.Step(equiweight_vector)

                X_next_t, wt_t, pf_value_t = state[0], state[1], state[2]
                pf_value_t_equiweight = state_equiweight[2]
                daily_returns_t = X_next_t[-1, :, -1]
                sharpe_ratio = round(sharpe_stocks(w=wt_previous),3)
                mdd = round(max_drawdown(weights=Wt_previous), 3)
                romad = round(RoMad(weights=Wt_previous, mdd=mdd))
                # print('------------------ testing -----------------------')
                # print('current portfolio value : ' + str(pf_value_previous))
                print('weights assigned : ' + str(wt_previous))
                print('sharpe_ratio:', sharpe_ratio)
                print('RoMad:', romad)
                # print('equiweight portfolio value : ' +␣
 ↪str(pf_value_t_equiweight))
                # print("equiweight sharpe", sharpe_equiweight)

                test_pf_values.append(pf_value_t)
                test_pf_values_equiweight.append(pf_value_t_equiweight)
                weight_vectors.append(wt_t)
                test_sharpe_ratio.append(sharpe_ratio)
                test_mdd.append(romad)

            print('------ test final value -------')
            print(test_pf_values)
            print(test_pf_values_equiweight)
            print(np.mean(test_sharpe_ratio))
            print(np.mean(test_mdd))
           # plot_cpv(test_pf_values, test_pf_values_equiweight,␣
 ↪portfolio_value_init_test, 'cpv_stocks.png')
            #plot_wts_assigned(weight_vectors[-1], ticker_num, ticker_list,␣
 ↪'wt_vector_stocks.png')


main()


toc=timeit.default_timer()
toc - tic #elapsed time in seconds
```

[14]:
```
0.0026616750000130196
```

[ ]:
```
#Below are the functions/programs called in previous main programs

# stock parameters


import numpy as np
import tensorflow as tf
import pandas as pd
```

```python
#dataset information
data_path = 'stocks_data_input.npy'
ticker_list = ['AAPL','AMZN','BA','GE','GOOG','JPM','MSFT']


data = np.load(data_path)
ohlc_features_num = data.shape[0]
ticker_num = data.shape[1]
trading_days_captured = data.shape[2]


print('number of ohlc features : ' + str(ohlc_features_num))
print('number of stocks considered : ' + str(ticker_num))
print('number of trading days captured : ' + str(trading_days_captured))

equiweight_weights_stocks = np.array([np.array([1/(ticker_num)]*(ticker_num+1))])


#hyper parameters of the CNN network
num_filters_layer_1 = 2
num_filters_layer_2 = 20
kernel_size = (1, 3)

#train-validation-test split
train_data_ratio = 0.6
training_steps = 0.6 * trading_days_captured
validation_steps = 0.2 * trading_days_captured
test_steps = 0.2 * trading_days_captured

#hyper parameters for RL framework
training_batch_size = 40
beta_pvm = 5e-5
num_trading_periods = 10

weight_vector_init = np.array(np.array([1] + [0] * ticker_num))
portfolio_value_init = 10000
weight_vector_init_test = np.array(np.array([1] + [0] * ticker_num))
portfolio_value_init_test = 10000
num_episodes = 4
num_batches = 40
equiweight_vector = np.array(np.array([1/(ticker_num + 1)] * (ticker_num + 1)))
#probability of exploitation in the RL framework (acting greedily)
epsilon = 0.9
#used while calculating the adjusted rewards
adjusted_rewards_alpha = 0.1
```

```
#hyper parameters for the optimizer
l2_reg_coef = 1e-8
adam_opt_alpha = 9e-2
#optimizer = tf.optimizers.Adam(adam_opt_alpha)

optimizer = tf.train.AdamOptimizer(adam_opt_alpha)

#hyper parameters for trading
trading_cost = 1/100000
interest_rate = 0.02/250
cash_bias_init = 0.7
```

[ ]:
```
#RL Environment
#Code Reference https://github.com/rathiromil13/
 ↪DS-5500-Project-Portfolio-Optimization-Using-Deep-Reinforcement-Learning



import numpy as np
import pandas as pd

class RLEnv():

    """
    Using this class we will render an RL trading environment

    PortfolioValue: value of the finance portfolio
    TransCost: Transaction cost that has to be paid by the agent to execute the␣
 ↪action
    ReturnRate: Percentage change in portfolio value
    WindowSize: Number of trading periods to be considered
    SplitSize: % of data to be used for training dataset, rest will be used for␣
 ↪test dataset
    """

    def __init__(self, Path, PortfolioValue, TransCost, ReturnRate, WindowSize,␣
 ↪TrainTestSplit):
        # Here, we need to initialize values like portfolio values, transaction␣
 ↪costs etc.

        # Loading dataset in Path to Dataset variable
        self.Dataset = np.load(Path)

        # Number of stocks and associated values like Close, High, Low
        self.NumStocks = self.Dataset.shape[1]
        self.NumValues = self.Dataset.shape[0]

        # Initializing parameter
        self.PortfolioValue = PortfolioValue
        self.TransCost = TransCost
        self.ReturnRate = ReturnRate
        self.WindowSize = WindowSize
        self.Done = False

        # Initiate state, action
        self.state = None
        self.TimeLength = None
        self.Terminate = False

        # Termination cutoff
        self.TerminateRows = int((self.Dataset.shape[2] - self.WindowSize) *␣
↪TrainTestSplit)

    def UpdatedOpenValues(self, T):
        # This function provides the
        return np.array([1+self.ReturnRate]+self.Dataset[-1,:,T].tolist())

    def InputTensor(self, Tensor, T):
        return Tensor[: , : , T - self.WindowSize:T]

    def ResetEnvironment(self, InitWeight, InitPortfolio, T):
        self.state= (self.InputTensor(self.Dataset, self.WindowSize) ,␣
↪InitWeight , InitPortfolio)
        self.TimeLength = self.WindowSize + T
        self.Done = False

        return self.state, self.Done

    def Step(self, Action):
        """
        Here, we get the action that needs to be performed at time step t, so,␣
↪we get new weight vector,
        reward function, updated value of portfolio

        We get the input tensor values for timestep t and for a given window␣
↪size for each of the stocks

        State usually contains a input tensor, weight vector, portfolio vector
        """

        # Obtain input tensor
        Dataset = self.InputTensor(self.Dataset, int(self.TimeLength))


        # Current state values - current weight vector and portfolio vector
        weight_vector_old = self.state[1]
        portfolio_value_old = self.state[2]

        # Update the vector with opening values
        NewOpenValues = self.UpdatedOpenValues(int(self.TimeLength))

        # Trading agent here provides new actions, that is new weight vector␣
↪using which new
        # allocations have to be done
        WeightAllocation = Action
        PortfolioAllocation = portfolio_value_old

        # While reallocating portfolios using weights we will have to account␣
↪for transaction or
        # commision rates
        TransactionCost = PortfolioAllocation * self.TransCost * np.linalg.
↪norm((WeightAllocation-weight_vector_old),ord = 1)

        # Inorder to find the new weight vector we need to obtain the value of␣
↪present portfolio
        # So as to obtain the value vector for each stock we need to multiply␣
↪the portfolio value with the weight vector
        # Every time a stock portfolio is updated there is an additional␣
↪transaction cost that incurs on the portfolio
        # value
        ValueAfterTransaction = (PortfolioAllocation * WeightAllocation) - np.
↪array([TransactionCost]+ [0] * self.NumStocks)

        # So the valueaftertransaction has cost deducted stock values for the␣
↪previous day, when we multiply this vector
        # with the latest open values
        NewValueofStocks = ValueAfterTransaction * NewOpenValues

        # When we sum the stock prices of individual stock prices, we get the␣
↪value of portfolio
        NewPortfolioValue = np.sum(NewValueofStocks)

        # Inorder to obtain the new weight vector, we divide individual stock␣
↪prices with total portfolio value
        NewWeightVector = NewValueofStocks / NewPortfolioValue

        # After each timestep, value of the portfolio either decreases or␣
↪increases depending on how the agent
        # performs
        RewardValue = (NewPortfolioValue - portfolio_value_old) /␣
↪(portfolio_value_old)
```

```
        self.TimeLength = self.TimeLength + 1

        # Using the computed values till now we can create new state
        self.state = (self.InputTensor(self.Dataset, int(self.TimeLength)),␣
 ↪NewWeightVector, NewPortfolioValue)

        # Here, we have to compute termination criteria, when to terminate the␣
 ↪step process
        if self.TimeLength >= self.TerminateRows:
            self.Done = True

        return self.state, RewardValue, self.Done
```

[ ]:
```
#PVM Setup

import numpy as np

class PFVectorMemory(object):

    def __init__(self, ticker_num, beta_pvm, training_steps,␣
 ↪training_batch_size, wt_vector_init):

        #pvm at all times
        self.pvm = np.transpose(np.array([wt_vector_init] * int(training_steps)))
        self.beta_pvm = beta_pvm
        self.training_steps = training_steps
        self.training_batch_size = training_batch_size

    def get_wt_vector_t(self, t):
        return self.pvm[:, t]

    def update_wt_vector_t(self, t, weight):
        self.pvm[:, int(t)] = weight

    def test(self):
        return self.pvm
```

[ ]:
```
#Define CNN


import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import numpy as np

class PolicyCNN(object):
    '''
    Using this class we will build policy for the cnn network.
    '''

    def __init__(self, ohlc_feature_num, ticker_num, num_trading_periods, sess,␣
↪optimizer, trading_cost, cash_bias_init, interest_rate,
        equiweight_vector, adjusted_rewards_alpha, kernel_size,␣
↪num_filter_layer_1, num_filter_layer_2):

        # parameters
        self.ohlc_feature_num = ohlc_feature_num
        self.ticker_num = ticker_num
        self. num_trading_periods =  num_trading_periods
        self.trading_cost = trading_cost
        self.cash_bias_init = cash_bias_init
        self.interest_rate = interest_rate
        self.equiweight_vector = equiweight_vector
        self.adjusted_rewards_alpha = adjusted_rewards_alpha
        self.kernel_size = kernel_size
        self.num_filter_layer_1 = num_filter_layer_1
        self.num_filter_layer_2 = num_filter_layer_2
        self.optimizer = optimizer
        self.sess = sess
        self.X_t = tf.placeholder(tf.float32, [None, self.ohlc_feature_num, self.
↪ticker_num, self.num_trading_periods])
        self.weights_previous_t = tf.placeholder(tf.float32, [None, self.
↪ticker_num + 1])
        self.pf_previous_t = tf.placeholder(tf.float32, [None, 1])
        self.daily_returns_t = tf.placeholder(tf.float32, [None, self.
↪ticker_num])
        cash_bias = tf.get_variable('cash_bias', shape=[1, 1, 1, 1], initializer␣
↪= tf.constant_initializer(self.cash_bias_init))
        shape_X_t = tf.shape(self.X_t)[0]
        self.cash_bias = tf.tile(cash_bias, tf.stack([shape_X_t, 1, 1, 1]))

        #test1 = tf.variable(self.X_t)
       # test2 = tf.variable(self.weights_previous_t)


        def convolution_layers(X_t, num_filter_layer_1, kernel_size,␣
↪num_filter_layer_2, num_trading_periods):
            with tf.variable_scope("Convolution1"):
                convolution1 = tf.layers.conv2d(
                inputs = tf.transpose(X_t, perm=[0, 3, 2, 1]),
                activation = tf.nn.tanh,
                filters = num_filter_layer_1,
                strides = (1, 1),
                kernel_size = kernel_size,
                padding = 'same')


            with tf.variable_scope("Convolution2"):
                convolution2 = tf.layers.conv2d(
                inputs = convolution1,
                activation = tf.nn.tanh,
                filters = num_filter_layer_2,
                strides = (num_trading_periods, 1),
                kernel_size = (1, num_trading_periods),
                padding = 'same')

            with tf.variable_scope("Convolution3"):
                self.convolution3 = tf.layers.conv2d(
                    inputs = convolution2,
                    activation = tf.nn.relu,
                    filters = 1,
                    strides = (num_filter_layer_2 + 1, 1),
                    kernel_size = (1, 1),
                    padding = 'same')

            return self.convolution3


        def policy_output(convolution, cash_bias):
            with tf.variable_scope("Policy-Output"):
                tensor_squeeze = tf.squeeze(tf.concat([cash_bias, convolution],␣
↪axis=2), [1, 3])
                self.action = tf.nn.softmax(tensor_squeeze)
            return self.action


        def reward(shape_X_t, action_chosen, interest_rate, weights_previous_t,␣
↪pf_previous_t, daily_returns_t, trading_cost):
            #Calculating reward for current Portfolio
            with tf.variable_scope("Reward"):
                cash_return = tf.tile(tf.constant(1 + interest_rate, shape=[1,␣
↪1]), tf.stack([shape_X_t, 1]))
                y_t = tf.concat([cash_return, daily_returns_t], axis=1)
                pf_vector_t = action_chosen * pf_previous_t
                pf_vector_previous = weights_previous_t * pf_previous_t

                total_trading_cost = trading_cost * tf.norm(pf_vector_t -␣
↪pf_vector_previous, ord=1, axis=1) * tf.constant(1.0, shape=[1])
                total_trading_cost = tf.expand_dims(total_trading_cost, 1)
                zero_vector = tf.tile(tf.constant(np.array([0.0] * ticker_num).
↪reshape(1, ticker_num), shape=[1, ticker_num], dtype=tf.float32), tf.
↪stack([shape_X_t, 1]))
                cost_vector = tf.concat([total_trading_cost, zero_vector],␣
↪axis=1)

                pf_vector_second_t = pf_vector_t - cost_vector
                final_pf_vector_t = tf.multiply(pf_vector_second_t, y_t)
                portfolio_value = tf.norm(final_pf_vector_t, ord=1)
                self.instantaneous_reward = (portfolio_value - pf_previous_t) /␣
↪pf_previous_t

            #Calculating Reward for Equiweight portfolio
            with tf.variable_scope("Reward-Equiweighted"):
                cash_return = tf.tile(tf.constant(1 + interest_rate, shape=[1,␣
↪1]), tf.stack([shape_X_t, 1]))
                y_t = tf.concat([cash_return, daily_returns_t], axis=1)

                pf_vector_eq = self.equiweight_vector * pf_previous_t

                portfolio_value_eq = tf.norm(tf.multiply(pf_vector_eq, y_t),␣
↪ord=1)
                self.instantaneous_reward_eq = (portfolio_value_eq -␣
↪pf_previous_t) / pf_previous_t

            #Calculating Adjusted Rewards
            with tf.variable_scope("Reward-adjusted"):
                self.adjusted_reward = self.instantaneous_reward - self.
↪instantaneous_reward_eq - self.adjusted_rewards_alpha * tf.
↪reduce_max(action_chosen)

            return self.adjusted_reward

        self.convolution = convolution_layers(self.X_t, self.num_filter_layer_1,␣
↪self.kernel_size, self.num_filter_layer_2, self.num_trading_periods)
        self.action_chosen = policy_output(self.convolution, self.cash_bias)
        self.adjusted_reward = reward(shape_X_t, self.action_chosen, self.
↪interest_rate, self.weights_previous_t, self.pf_previous_t, self.
↪daily_returns_t, self.trading_cost)
        self.train_op = optimizer.minimize(-self.adjusted_reward)# added var_list

    def compute_weights(self, X_t_, weights_previous_t_):
        return self.sess.run(tf.squeeze(self.action_chosen), feed_dict={self.X_t:
↪ X_t_, self.weights_previous_t: weights_previous_t_})
```

```
    def train_cnn(self, X_t_, weights_previous_t_, pf_previous_t_,␣
↪daily_returns_t_):
        """
        training the neural network
        """
        self.sess.run(self.train_op, feed_dict={self.X_t: X_t_,
                                                self.weights_previous_t:␣
↪weights_previous_t_,
                                                self.pf_previous_t:␣
↪pf_previous_t_,
                                                self.daily_returns_t:␣
↪daily_returns_t_})
```

[ ]:
```
# Define LSTM

from __future__ import print_function
import tensorflow as tf
import numpy as np
import tflearn

class PolicyLSTM(object):
    '''
    Using this class we will build policy for the LSTM network.
    '''

    def __init__(self, ohlc_feature_num, ticker_num, num_trading_periods, sess,␣
↪optimizer, trading_cost, cash_bias_init, interest_rate, equiweight_vector,␣
↪adjusted_rewards_alpha, num_filter_layer):

        # parameters
        self.ohlc_feature_num = ohlc_feature_num
        self.ticker_num = ticker_num
        self.num_trading_periods =  num_trading_periods
        self.trading_cost = trading_cost
        self.cash_bias_init = cash_bias_init
        self.interest_rate = interest_rate
        self.equiweight_vector = equiweight_vector
        self.adjusted_rewards_alpha = adjusted_rewards_alpha
        self.optimizer = optimizer
        self.sess = sess
        self.num_filter_layer = num_filter_layer
        layers=2
        lstm_neurons = 20

        self.X_t = tf.placeholder(tf.float32, [None, self.ohlc_feature_num, self.
↪ticker_num, self.num_trading_periods])
```

```
        self.weights_previous_t = tf.placeholder(tf.float32, [None, self.
↪ticker_num + 1])
        self.pf_previous_t = tf.placeholder(tf.float32, [None, 1])
        self.daily_returns_t = tf.placeholder(tf.float32, [None, self.
↪ticker_num])
        cash_bias = tf.get_variable('cash_bias', shape=[1, 1, 1, 1], initializer␣
↪= tf.constant_initializer(self.cash_bias_init))
        shape_X_t = tf.shape(self.X_t)[0]
        self.cash_bias = tf.tile(cash_bias, tf.stack([shape_X_t, 1, 1, 1]))


        def lstm(X_t):
            network = tf.transpose(X_t, [0, 1, 3, 2])
            network = network / network[:, :, -1, 0, None, None]

            for layer_number in range(layers):
                resultlist = []
                reuse = False

                for i in range(self.ticker_num):
                    if i > 0:
                        reuse = True

                    result = tflearn.layers.lstm(X_t[:,:,:, i],
                                                    lstm_neurons,
                                                    dropout=0.3,
                                                   ␣
↪scope="lstm"+str(layer_number),
                                                    reuse=reuse)


                    resultlist.append(result)
                network = tf.stack(resultlist)
                network = tf.transpose(network, [1, 0, 2])
                network = tf.reshape(network, [-1, 1, self.ticker_num,␣
↪lstm_neurons])
                # print('dhsegfhebgfhewf', network.shape)
            return network

        def policy_output(network, cash_bias):
            with tf.variable_scope("Convolution_Layer"):
                self.conv = tf.layers.conv2d(
                    inputs = network,
                    activation = tf.nn.relu,
                    filters = 1,
                    strides = (num_filter_layer + 1, 1),
                    kernel_size = (1, 1),
                    padding = 'same')

            with tf.variable_scope("Policy-Output"):
                tensor_squeeze = tf.squeeze(tf.concat([cash_bias, self.conv],␣
↪axis=2), [1,3])
                self.action = tf.nn.softmax(tensor_squeeze)
            return self.action


        def reward(shape_X_t, action_chosen, interest_rate, weights_previous_t,␣
↪pf_previous_t, daily_returns_t, trading_cost):
            #Calculating reward for current Portfolio
            with tf.variable_scope("Reward"):
                cash_return = tf.tile(tf.constant(1 + interest_rate, shape=[1,␣
↪1]), tf.stack([shape_X_t, 1]))
                y_t = tf.concat([cash_return, daily_returns_t], axis=1)
                pf_vector_t = action_chosen * pf_previous_t
                pf_vector_previous = weights_previous_t * pf_previous_t

                total_trading_cost = trading_cost * tf.norm(pf_vector_t -␣
↪pf_vector_previous, ord=1, axis=1) * tf.constant(1.0, shape=[1])
                total_trading_cost = tf.expand_dims(total_trading_cost, 1)

                zero_vector = tf.tile(tf.constant(np.array([0.0] * ticker_num).
↪reshape(1, ticker_num), shape=[1, ticker_num], dtype=tf.float32), tf.
↪stack([shape_X_t, 1]))
                cost_vector = tf.concat([total_trading_cost, zero_vector],␣
↪axis=1)


                pf_vector_second_t = pf_vector_t - cost_vector
                final_pf_vector_t = tf.multiply(pf_vector_second_t, y_t)
                portfolio_value = tf.norm(final_pf_vector_t, ord=1)
                self.instantaneous_reward = (portfolio_value - pf_previous_t) /␣
↪pf_previous_t

            #Calculating Reward for Equiweight portfolio
            with tf.variable_scope("Reward-Equiweighted"):
                cash_return = tf.tile(tf.constant(1 + interest_rate, shape=[1,␣
↪1]), tf.stack([shape_X_t, 1]))
                y_t = tf.concat([cash_return, daily_returns_t], axis=1)

                pf_vector_eq = self.equiweight_vector * pf_previous_t

                portfolio_value_eq = tf.norm(tf.multiply(pf_vector_eq, y_t),␣
↪ord=1)
```

```
                self.instantaneous_reward_eq = (portfolio_value_eq -␣
↪pf_previous_t) / pf_previous_t

            #Calculating Adjusted Rewards
            with tf.variable_scope("Reward-adjusted"):
                self.adjusted_reward = self.instantaneous_reward - self.
↪instantaneous_reward_eq - self.adjusted_rewards_alpha * tf.
↪reduce_max(action_chosen)

            return self.adjusted_reward


        self.lstm_layer = lstm(self.X_t)
        self.action_chosen = policy_output(self.lstm_layer, self.cash_bias)
        self.adjusted_reward = reward(shape_X_t, self.action_chosen, self.
↪interest_rate, self.weights_previous_t, self.pf_previous_t, self.
↪daily_returns_t, self.trading_cost)
        self.train_op = optimizer.minimize(-self.adjusted_reward)

    def compute_weights(self, X_t_, weights_previous_t_):
        # tf.print(self.action_chosen)
        return self.sess.run(tf.squeeze(self.action_chosen), feed_dict={self.X_t:
↪ X_t_, self.weights_previous_t: weights_previous_t_})

    def train_lstm(self, X_t_, weights_previous_t_, pf_previous_t_,␣
↪daily_returns_t_):
        """
        training the neural network
        """
        self.sess.run(self.train_op, feed_dict={self.X_t: X_t_,
                                                self.weights_previous_t:␣
↪weights_previous_t_,
                                                self.pf_previous_t:␣
↪pf_previous_t_,
                                                self.daily_returns_t:␣
↪daily_returns_t_})
```

Reference

[1] Heaton, J.B. and Polson, Nick and Witte, Jan, Deep Learning for Finance: Deep Portfolios (September 5, 2016). Applied Stochastic Models in Business and Industry 33 (1), 3-12. Available at SSRN: https://ssrn.com/abstract=2838013 or http://dx.doi.org/10.2139/ssrn.2838013

[2] K. Nakagawa, T. Uchida, and T. Aoshima. Deep Factor Model -- Explaining Deep Learning Decisions for Forecasting Stock Returns with Layer-wise Relevance Propagation. ECML PKDD Workshops, 2018. arXiv:1810.01278

[3] James Cumming. An Investigation into the Use of Reinforcement Learning Techniques within the Algorithmic Trading Domain. Retrieved from https://www.doc.ic.ac.uk/teaching/distinguished-projects/2015/j.cumming.pdf

[4] Nitin Kanwar. Deep Reinforcement Learning-based Portfolio Management (2019 May).URL http://hdl.handle.net/10106/28108s

[5] Ruohan Zhan, Tianchang He and Yunpo Li. Deep Reinforcement Learning in Portfolio Management. Retrieved from http://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/th7/poster.pdf

[6] H. Sak, A. Senior, and F. Beaufays. Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling (2014). arXiv:1402.1128

[7]. Zhengyao Jiang. Cryptocurrency Portfolio Management with Deep Reinforcement Learning (2016 December). arXiv:1612.01277

[8] Zhengyao Jiang, Dixing Xu and Jinjun Liang. A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem (2017 June). arXiv:1706.10059

[9] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning (2015 September). arXiv:1509.02971

[10] Zhipeng Lian, Hao Chen, Junhao Zhu, Kangkang Jiang, Yanran Li. Adversarial Deep Reinforcement Learning in Portfolio Management (2018 November). arXiv:1808.09940v3

[11] Diederik P. Kingm and Jimmy Lei Ba. Adam: A Method for Stochastic Optimization (2014 December). arXiv:1412.6980